

HOMEWORK 1&2

Integrazione e Test di Sistemi Software

SNTDNT

Donato Santacesaria

Matricola 754727

Homework 1

1: Realizzare Specification-based testing

Approccio a 7 step

Step 1: Comprendere i requisiti

Cosa il programma **dovrebbe fare** e come gli **input** sono convertiti in **output** attesi.

```
public static int[] generateUniqueRandomNumbersInRange(String lowerLimit, String upperLimit, String numCount)
```

1. **Obiettivo:** generare un array di numeri casuali unici all'interno di un intervallo specificato.
2. **Input:** tre parametri rappresentati come stringhe:
 1. 'lowerLimit': limite inferiore (incluso) dell'intervallo;
 2. 'upperLimit': limite superiore (incluso) dell'intervallo;
 3. 'numCount': numero di elementi da generare nell'array.

Le stringhe di input vengono convertite in interi.

3. **Output:** il programma restituisce un array ordinato di interi contenenti numeri casuali unici all'interno dell'intervallo specificato. Se si verificano errori il programma lancerà un'eccezione con messaggio descrittivo dell'errore.

Step 1: Comprendere i requisiti

Cosa il programma **dovrebbe fare** e come gli **input** sono convertiti in **output** attesi.

```
public static int[] generateUniqueRandomNumbersInRange(String lowerLimit, String  
upperLimit, String numCount)
```

FOCUS_1 – CONVERSIONE DEGLI INPUT IN INTERI

Il metodo converte le stringhe di input in interi, gestendo la possibilità di errori di conversione, attraverso il lancio di un'eccezione.

Step 1: Comprendere i requisiti

Cosa il programma **dovrebbe fare** e come gli **input** sono convertiti in **output** attesi.

```
public static int[] generateUniqueRandomNumbersInRange(String lowerLimit, String upperLimit, String numCount)
```

FOCUS_2 – VERIFICA DEGLI INPUT

1. Il metodo verifica della **validità dell'intervallo** specificato: se il limite inferiore è maggiore uguale al limite superiore, viene lanciata un'eccezione;
2. Il metodo verifica se il **numero di elementi da generare è positivo**: se il numero di elementi non è positivo, viene sollevata un'eccezione;
3. Il metodo verifica se il **numero di elementi da generare è al massimo** pari alla differenza più uno tra il limite superiore e il limite inferiore: se il numero di elementi è superiore alle dimensioni dell'intervallo, viene sollevata un'eccezione.

Step 1: Comprendere i requisiti

Cosa il programma **dovrebbe fare** e come gli **input** sono convertiti in **output** attesi.

```
public static int[] generateUniqueRandomNumbersInRange(String lowerLimit, String upperLimit, String numCount)
```

FOCUS_3 – GENERAZIONE DEI NUMERI CASUALI

Il metodo **genera numeri interi casuali**, assicurandone l'**unicità**, fino a che non ne sono stati generati quanti indicati.

Quindi li inserisce in un **array ordinato**, in **ordine crescente**.

Il metodo **restituisce l'array** di numeri generati.

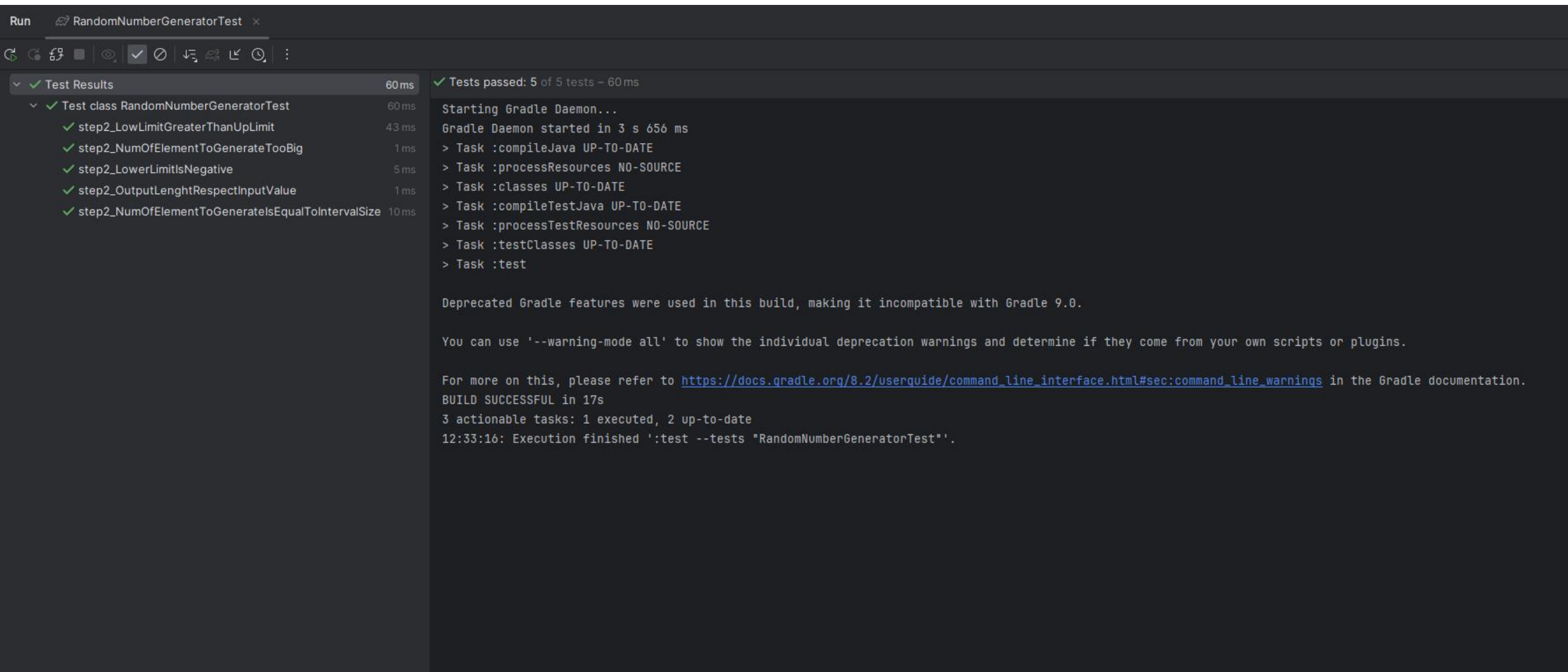
Step 2: Esplorare cosa fa il programma per vari input

```
1
2  import org.junit.jupiter.api.Test;
3  import static org.junit.jupiter.api.Assertions.*;
4  import org.example.RandomNumberGenerator;
5  new *
6  public class RandomNumberGeneratorTest {
7  new *
8      @Test
9      void step2_OutputLengthRespectInputValue(){
10         int[] result1 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "1", upperLimit: "10", numCount: "5");
11         assertEquals( expected: 5, result1.length);
12     }
13 new *
14     @Test
15     void step2_NumOfElementToGenerateIsEqualToIntervalSize(){
16         int[] result1 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "5", upperLimit: "7", numCount: "3");
17         assertArrayEquals(new int[]{5, 6, 7}, result1);
18     }
19 new *
20     @Test
21     void step2_LowerLimitIsNegative(){
22         int[] result1 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "-10", upperLimit: "10", numCount: "10");
23         assertEquals( expected: 10, result1.length);
24     }
25 new *
26     @Test
27     void step2_LowLimitGreaterThanOrEqualToUpLimit(){
```


Step 2: Esplorare cosa fa il programma per vari input

```
10     }
11     new *
12     @Test
13     void step2_NumOfElementToGenerateIsEqualToIntervalSize(){
14         int[] result1 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "5", upperLimit: "7", numCount: "3");
15         assertEquals(new int[]{5, 6, 7}, result1);
16     }
17     new *
18     @Test
19     void step2_LowerLimitIsNegative(){
20         int[] result1 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "-10", upperLimit: "10", numCount: "10");
21         assertEquals(expected: 10, result1.length);
22     }
23     new *
24     @Test
25     void step2_LowLimitGreaterThanOrEqualToUpLimit(){
26         assertThrows(IllegalArgumentException.class, () ->
27             RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "1", upperLimit: "10", numCount: "0"));
28     }
29     new *
30     @Test
31     void step2_NumOfElementToGenerateTooBig(){
32         assertThrows(IllegalArgumentException.class, () ->
33             RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "1", upperLimit: "10", numCount: "12"));
34     }
35 }
```

Step 2: Esplorare cosa fa il programma per vari input



The screenshot shows an IDE interface with a 'Run' tab at the top. Below the tab, there's a toolbar with icons for running, debugging, and other actions. The main area is divided into two panels. The left panel, titled 'Test Results', shows a tree view of test results for 'RandomNumberGeneratorTest'. The right panel shows the output of the Gradle build, including task execution details and deprecation warnings.

Test Results:

Test Results	Duration
Test class RandomNumberGeneratorTest	60 ms
step2_LowLimitGreaterThanUpLimit	43 ms
step2_NumOfElementToGenerateTooBig	1 ms
step2_LowerLimitIsNegative	5 ms
step2_OutputLenghtRespectInputValue	1 ms
step2_NumOfElementToGeneratelsEqualToIntervalSize	10 ms

Gradle Build Output:

```
Starting Gradle Daemon...
Gradle Daemon started in 3 s 656 ms
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava UP-TO-DATE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.2/userguide/command\_line\_interface.html#sec:command\_line\_warnings in the Gradle documentation.
BUILD SUCCESSFUL in 17s
3 actionable tasks: 1 executed, 2 up-to-date
12:33:16: Execution finished ':test --tests "RandomNumberGeneratorTest"'.
```

Step 3: Esplorare input, output e identificare partizioni

Esplorare:

A – SINGOLI INPUT (CLASSI DI INPUT)

lowerLimit

1. NULL string
2. Empty string
3. String to int
4. Not to int
(any string)

numCount

1. NULL string
2. Empty string
3. String to int
4. Not to int
(any string)

upperLimit

1. NULL string
2. Empty string
3. String to int
4. Not to int
(any string)

Step 3: Esplorare input, output e identificare partizioni

Esplorare:

B – COMBINAZIONI DI INPUT

1	La funzione riceve	0 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
2	La funzione riceve	0 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
3	La funzione riceve	0 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
4	La funzione riceve	0 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare non valido
5	La funzione riceve	1 elemento convertibile in Int	con intervallo valido	ed un numero di elementi da generare valido
6	La funzione riceve	1 elemento convertibile in Int	con intervallo valido	ed un numero di elementi da generare non valido
7	La funzione riceve	1 elemento convertibile in Int	con intervallo non valido	ed un numero di elementi da generare valido
8	La funzione riceve	1 elemento convertibile in Int	con intervallo non valido	ed un numero di elementi da generare non valido
9	La funzione riceve	2 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
10	La funzione riceve	2 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
11	La funzione riceve	2 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
12	La funzione riceve	2 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare non valido
13	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
14	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
15	La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
16	La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare non valido

Step 3: Esplorare input, output e identificare partizioni

Esplorare:

C – CLASSI DI OUTPUT (ATTESI)

Array di interi

1. Un solo elemento
2. Elementi multipli
...di cui ogni singolo elemento
 1. Int

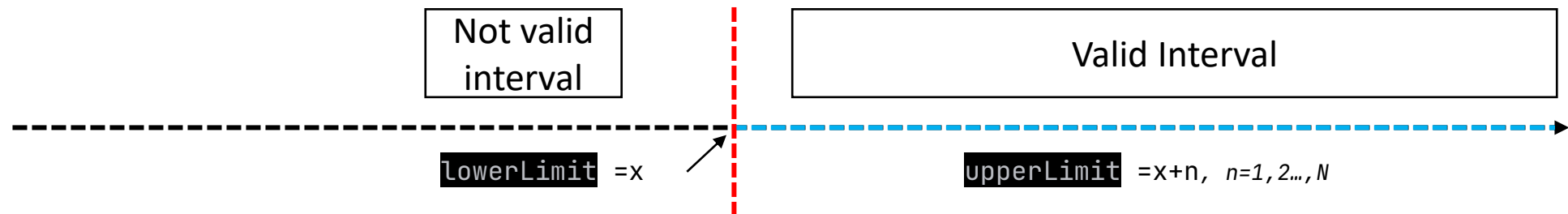
Eccezione

1. Intervallo non valido
2. Numero di elementi da generare non positivo
3. Conversione to Int fallita

Step 4: Identificare boudary cases (*aka* corner cases)

A) Test riguardanti la validità dell'intervallo

Intervallo valido \Leftrightarrow `lowerLimit` < `upperLimit`

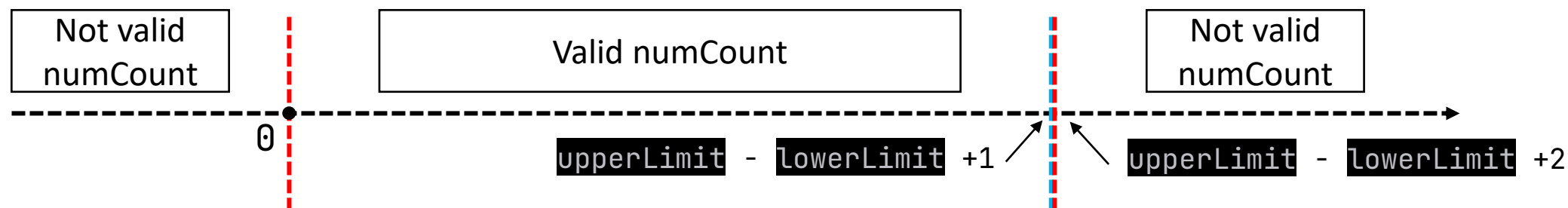


13 La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
15 La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido

Step 4: Identificare boudary cases (*aka* corner cases)

B) Test riguardanti la validità del numero di elementi da generare

Conteggio valido $\Leftrightarrow 1 \leq \text{numCount} \leq (\text{upperLimit} - \text{lowerLimit} + 1)$



13 La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
-----------------------	--------------------------------	-----------------------	---

14 La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
-----------------------	--------------------------------	-----------------------	---

Step 5: Elaborare casi di test

Decidere pragmaticamente quali partizioni dovrebbero essere combinate con altre e quali no

numCount

1. NULL string

2. Empty string

3. String to int

4. Not to int

(any string)

upperLimit

1. NULL string

2. Empty string

3. String to int

4. Not to int

(any string)

lowerLimit

1. NULL string

2. Empty string

3. String to int

4. Not to int

(any string)

1	La funzione riceve	0 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
2	La funzione riceve	0 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
3	La funzione riceve	0 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
4	La funzione riceve	0 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare non valido
5	La funzione riceve	1 elemento convertibile in Int	con intervallo valido	ed un numero di elementi da generare valido
6	La funzione riceve	1 elemento convertibile in Int	con intervallo valido	ed un numero di elementi da generare non valido
7	La funzione riceve	1 elemento convertibile in Int	con intervallo non valido	ed un numero di elementi da generare valido
8	La funzione riceve	1 elemento convertibile in Int	con intervallo non valido	ed un numero di elementi da generare non valido
9	La funzione riceve	2 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
10	La funzione riceve	2 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
11	La funzione riceve	2 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
12	La funzione riceve	2 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare non valido
13	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
14	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
15	La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
16	La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare non valido

Combinando tutti i possibili input: 4 x 4 x 4 x 16 = 1024 test.

Step 5: Elaborare casi di test

Decidere pragmaticamente quali partizioni dovrebbero essere combinate con altre e quali no

A) Testare casi eccezionali solo una volta e non combinarli (es. null, empty):

T1: `lowerLimit` è null.

T2: `lowerLimit` è empty.

T3: `upperLimit` è null.

T4: `upperLimit` è empty.

T5: `numCount` è null.

T6: `numCount` è empty.

Step 5: Elaborare casi di test

Decidere pragmaticamente quali partizioni dovrebbero essere combinate con altre e quali no

B) Una prima combinazione di input:

La funzione riceve 1 elemento convertibile in int

T7: **lowerLimit** è *Not To Int*, **upperLimit** è *Not To Int*, **CountNum** è *String To Int*.

T8: **lowerLimit** è *Not To Int*, **upperLimit** è *String To Int*, **CountNum** è *Not To Int*.

T9: **lowerLimit** è *String To Int*, **upperLimit** è *Not To Int*, **CountNum** è *Not To Int*.

Step 5: Elaborare casi di test

Decidere pragmaticamente quali partizioni dovrebbero essere combinate con altre e quali no

C) Una seconda combinazione di input:

La funzione riceve 2 elementi convertibili in int

T10: `lowerLimit` è *Not To Int*, `upperLimit` è *String To Int*, `CountNum` è *String To Int*.

T11: `lowerLimit` è *String To Int*, `upperLimit` è *Not To Int*, `CountNum` è *String To Int*.

T12: `lowerLimit` è *String To Int*, `upperLimit` è *String To Int*, `CountNum` è *Not To Int*.

Step 5: Elaborare casi di test

Decidere pragmaticamente quali partizioni dovrebbero essere combinate con altre e quali no

D) Una terza combinazione di input:

La funzione riceve 3 elementi convertibili in int

T#: combinazioni coincidenti con i casi coperti dai **boundary cases**:

13	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
15	La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
14	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido

(seguono **boundary test**...)

Step 5: Elaborare casi di test

Decidere pragmaticamente quali partizioni dovrebbero essere combinate con altre e quali no

E) Boundary test:

13	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare valido
T13:	$\text{lowerLimit} < \text{upperLimit}, 1 \leq \text{numCount} \leq (\text{upperLimit} - \text{lowerLimit} + 1)$			
T14:	$\text{lowerLimit} < \text{upperLimit}, \text{numCount} = (\text{upperLimit} - \text{lowerLimit} + 1)$			
T15:	$\text{lowerLimit} < \text{upperLimit}, \text{numCount} = 1$			
15	La funzione riceve	3 elementi convertibili in Int	con intervallo non valido	ed un numero di elementi da generare valido
T16:	$\text{lowerLimit} > \text{upperLimit}, 1 \leq \text{numCount} \leq (\text{upperLimit} - \text{lowerLimit} + 1)$			
T17:	$\text{lowerLimit} = \text{upperLimit}, 1 \leq \text{numCount} \leq (\text{upperLimit} - \text{lowerLimit} + 1)$			
14	La funzione riceve	3 elementi convertibili in Int	con intervallo valido	ed un numero di elementi da generare non valido
T18:	$\text{lowerLimit} < \text{upperLimit}, \text{numCount} < 0$			
T19:	$\text{lowerLimit} < \text{upperLimit}, \text{numCount} = 0$			
T20:	$\text{lowerLimit} < \text{upperLimit}, \text{numCount} > (\text{upperLimit} - \text{lowerLimit} + 1)$			

Nota finale: chiudiamo con 20 test al posto di 1024.

Step 6: Automatizzare casi di test

T1: `lowerLimit` è null.

T2: `lowerLimit` è empty.

T3: `upperLimit` è null.

T4: `upperLimit` è empty.

T5: `numCount` è null.

T6: `numCount` è empty.

```
new *
public class RandomNumberGeneratorAutomateTestCase {
    new *
    @ParameterizedTest
    @CsvSource(value = {
        "LOW LIMIT, UP LIMIT, #ELEMENTS",
        "null,10,5", //T1: lowerLimit è null
        ",10,5", //T2: lowerLimit è empty
        "1,null,5", //T3: upperLimit è null
        "1,,5", //T4: upperLimit è empty
        "1,10,null", //T5: numCount è null
        "1,10,", //T6: numCount è empty
    }, useHeadersInDisplayName = true)
    void oneElementNullOrEmptyThrowsException(String lower, String upper, String numCount) {
        assertThrows(IllegalArgumentException.class, () ->
            RandomNumberGenerator.generateUniqueRandomNumbersInRange(lower, upper, numCount));
    }
}
```

Step 6: Automatizzare casi di test

- T7: `lowerLimit` è *Not To Int*, `upperLimit` è *Not To Int*, `numCount` è *String To Int*.
T8: `lowerLimit` è *Not To Int*, `upperLimit` è *String To Int*, `numCount` è *Not To Int*.
T9: `lowerLimit` è *String To Int*, `upperLimit` è *Not To Int*, `numCount` è *Not To Int*.

```
new *
@ParameterizedTest
@CsvSource(value = {
    "LOW LIMIT, UP LIMIT, #ELEMENTS",
    "a,2147483648,5", // T7: solo numCount è String To Int
    "a,10,%",        // T8: solo upperLimit è String To Int
    "1,2147483648,@", // T9: solo lowerLimit è String To Int
}, useHeadersInDisplayName = true)
void oneElementStringToIntThrowsException(String lower, String upper, String numCount) {
    assertThrows(IllegalArgumentException.class, () ->
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(lower, upper, numCount));
}
```

Step 6: Automatizzare casi di test

- T10: `lowerLimit` è *Not To Int*, `upperLimit` è *String To Int*, `numCount` è *String To Int*.
T11: `lowerLimit` è *String To Int*, `upperLimit` è *Not To Int*, `numCount` è *String To Int*.
T12: `lowerLimit` è *String To Int*, `upperLimit` è *String To Int*, `numCount` è *Not To Int*.

```
new *
@ParameterizedTest
@CsvSource(value = {
    "LOW LIMIT, UP LIMIT, #ELEMENTS",
    "a,10,5",           // T10: lowerLimit è Not To Int, upperLimit è String To Int, numCount è String To Int.
    "1,2147483648,5",   // T11: lowerLimit è String To Int, upperLimit è Not To Int, numCount è String To Int.
    "1,10,@",           // T12: lowerLimit è String To Int, upperLimit è String To Int, numCount è Not To Int.
}, useHeadersInDisplayName = true)
void twoElementStringToIntThrowsException(String lower, String upper, String numCount) {
    assertThrows(IllegalArgumentException.class, () ->
        | RandomNumberGenerator.generateUniqueRandomNumbersInRange(lower, upper, numCount));
}
```


Step 6: Automatizzare casi di test

T13: `lowerLimit < upperLimit, 1 ≤ numCount ≤ (upperLimit - lowerLimit + 1)`

T14: `lowerLimit < upperLimit, numCount = (upperLimit - lowerLimit + 1)`

Nota: il T14 è utile anche per verificare l'ordinamento crescente dell'output e l'unicità degli elementi.

T15: `lowerLimit < upperLimit, numCount = 1`

```
new *
@Test
void threeElementsStringToIntWithValidIntervalAndValidNumCountReturnSortedArray(){
    int[] result1 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "1", upperLimit: "10", numCount: "5");
    assertEquals( expected: 5, result1.length); //T13: lowerLimit < upperLimit, 1 ≤ numCount ≤ (upperLimit - lowerLimit + 1)
    int[] result2 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "5", upperLimit: "7", numCount: "3");
    assertEquals(new int[]{5, 6, 7}, result2); //T14: lowerLimit < upperLimit, numCount = (upperLimit - lowerLimit + 1)
    int[] result3 = RandomNumberGenerator.generateUniqueRandomNumbersInRange( lowerLimit: "1", upperLimit: "10", numCount: "1");
    assertEquals( expected: 1, result3.length); //T15: lowerLimit < upperLimit, numCount = 1
}
```

Step 6: Automatizzare casi di test

T16: `lowerLimit > upperLimit`, $1 \leq \text{numCount} \leq (\text{upperLimit} - \text{lowerLimit} + 1)$

T17: `lowerLimit = upperLimit`, $1 \leq \text{numCount} \leq (\text{upperLimit} - \text{lowerLimit} + 1)$

```
new *
@ParameterizedTest
@CsvSource(value = {
    "LOW LIMIT, UP LIMIT, #ELEMENTS",
    "10,1,5",    // T16: lowerLimit > upperLimit, 1 ≤ numCount ≤ (upperLimit - lowerLimit + 1)
    "10,10,1"    // T17: lowerLimit = upperLimit, 1 ≤ numCount ≤ (upperLimit - lowerLimit + 1)
}, useHeadersInDisplayName = true)
void threeElementsStringToIntWithNOTValidIntervalAndValidNumCountThrowsException(String lower, String upper, String numCount) {
    assertThrows(IllegalArgumentException.class, () ->
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(lower, upper, numCount));
}
```

Step 6: Automatizzare casi di test

T18: `lowerLimit < upperLimit, numCount < 0`

T19: `lowerLimit < upperLimit, numCount = 0`

T20: `lowerLimit < upperLimit, numCount > (upperLimit - lowerLimit + 1)`

```
@ParameterizedTest
@CsvSource(value = {
    "LOW LIMIT, UP LIMIT, #ELEMENTS",
    "1,10,-5", // T18: lowerLimit < upperLimit, numCount < 0
    "1,10,0",  // T19: lowerLimit < upperLimit, numCount = 0
    "1,10,11"  // T20: lowerLimit < upperLimit, numCount > (upperLimit - lowerLimit + 1)
}, useHeadersInDisplayName = true)
void threeElementsStringToIntWithValidIntervalAndNOTValidNumCountThrowsException(String lower, String upper, String numCount) {
    assertThrows(IllegalArgumentException.class, () ->
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(lower, upper, numCount));
}
```

Step 7: Aumenta la suite con creatività ed esperienza

In condizioni di validità (i.e. limite valido e valido numero di elementi da generare nell'intervallo), uno dei tre elementi della funzione contiene uno whitespace:

T21: `lowerLimit` è *String To Int + Whitespace*, `upperLimit` è *String To Int*, `CountNum` è *String To Int*.

T22: `lowerLimit` è *String To Int*, `upperLimit` è *String To Int + Whitespace*, `CountNum` è *String To Int*.

T23: `lowerLimit` è *String To Int*, `upperLimit` è *String To Int*, `CountNum` è *String To Int + Whitespace*.

```
new *
@ParameterizedTest
@CsvSource(value = {
    "LOW LIMIT, UP LIMIT, #ELEMENTS",
    "' 1',10,5",    // T21: lowerLimit è String To Int + Whitespace, upperLimit è String To Int, CountNum è String To Int.
    "1,' 10',5",    // T22: lowerLimit è String To Int, upperLimit è String To Int + Whitespace, CountNum è String To Int.
    "1,10,' 5'"     // T23: lowerLimit è String To Int, upperLimit è String To Int, CountNum è String To Int + Whitespace.
}, useHeadersInDisplayName = true)
void whiteSpaceInOneElementThrowsException(String lower, String upper, String numCount) {
    assertThrows(IllegalArgumentException.class, () ->
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(lower, upper, numCount));
}
```

Nota: altre tipologie di caratteri speciali sono stati utilizzati negli altri test.

2: Leggere l'implementazione

Nel caso in cui il codice non sia stato scritto personalmente

```
package org.example;
```

```
// Press Shift twice to open the Search Everywhere dialog and type `show whitespaces`,  
// then press Enter. You can now see whitespace characters in your code.
```

```
import java.util.Random;  
import java.util.Scanner;  
import java.util.*;
```

```
7 usages  donato_santacesaria *
```

```
public class RandomNumberGenerator {
```

```
6 usages  donato_santacesaria *
```

```
public static int[] generateUniqueRandomNumbersInRange(String lowerLimit, String upperLimit, String numCount)  
    throws IllegalArgumentException {
```

```
    try {
```

```
        // Conversione degli operandi in interi  
        int lower = Integer.parseInt(lowerLimit);  
        int upper = Integer.parseInt(upperLimit);  
        int count = Integer.parseInt(numCount);
```

```
        if((upper-lower+1)<count){
```

```
            throw new IllegalArgumentException("\n->Errore: Il numero degli elementi da generare deve essere al massimo pari alla differenza+1 tra il limite superiore ed il  
        }
```

```
        // Verifica se l'intervallo è valido
```

```
        if (lower >= upper) {
```

```
            throw new IllegalArgumentException("\n->Errore: Limite inferiore deve essere inferiore al limite superiore.");  
        }
```

```
        // Verifica se il numero di numeri è positivo
```

```
        if (count <= 0) {
```

```
            throw new IllegalArgumentException("\n->Errore: Il numero di elementi deve essere positivo.");  
        }
```

```
        Set<Integer> uniqueNumbers = new HashSet<>();
```

```
        Random random = new Random();
```

```

16 int lower = Integer.parseInt(lowerLimit);
17 int upper = Integer.parseInt(upperLimit);
18 int count = Integer.parseInt(numCount);
19
20 if((upper-lower+1)<count){
21     throw new IllegalArgumentException("\n->Errore: Il numero degli elementi da generare deve essere al massimo pari alla differenza+1 tra il limite superiore ed il
22 }
23
24 // Verifica se l'intervallo è valido
25 if (lower >= upper) {
26     throw new IllegalArgumentException("\n->Errore: Limite inferiore deve essere inferiore al limite superiore.");
27 }
28
29 // Verifica se il numero di numeri è positivo
30 if (count <= 0) {
31     throw new IllegalArgumentException("\n->Errore: Il numero di elementi deve essere positivo.");
32 }
33
34 Set<Integer> uniqueNumbers = new HashSet<>();
35 Random random = new Random();
36
37 // Genera numeri unici nell'intervallo
38 while (uniqueNumbers.size() < count) {
39     int randomNumber = random.nextInt( bound: upper - lower + 1) + lower;
40     uniqueNumbers.add(randomNumber);
41 }
42
43 // Converti il set in un array ordinato
44 int[] result = uniqueNumbers.stream().sorted().mapToInt(Integer::intValue).toArray();
45
46 return result;
47
48 } catch (NumberFormatException e) {
49     throw new IllegalArgumentException("\n->Errore: Inserito un valore non valido. Assicurati di inserire valori validi.");
50 }
51 }
52 }
53

```

3: Eseguire test con tool di code coverage

Per identificare in modo automatico parti non coperte

project

RandomNumberGenerator [untitled1]

C:\Users\donat\IdeaProjects\RandomNumberGe

.gradle

.idea

build

classes

generated

reports

test-results

tmp

gradle

htmlReport

src

main

java

100% classes, 100% lines covered

org.example

100% classes, 100% lines covered

RandomNumberGenerator

100% methods, 100% lines covered

resources

test

java

RandomNumberGeneratorAutomateTestCase

resources

.gitignore

build.gradle.kts

gradlew

gradlew.bat

settings.gradle.kts

External Libraries

Scratches and Consoles

RandomNumberGenerator.java

RandomNumberGeneratorAutomateTestCase.java

2

50

package org.example;

// Press Shift twice to open the Search Everywhere dialog and type `show whites

// then press Enter. You can now see whitespace characters in your code.

import java.util.Random;

import java.util.Scanner;

import java.util.*;

donato_santacesaria *

public class RandomNumberGenerator {

9 usages donato_santacesaria *

public static int[] generateUniqueRandomNumbersInRange(String lowerLimit,

throws IllegalArgumentException {

try {

// Conversione degli operandi in interi

int lower = Integer.parseInt(lowerLimit);

int upper = Integer.parseInt(upperLimit);

int count = Integer.parseInt(numCount);

if((upper-lower+1)<count){

throw new IllegalArgumentException("\n->Errore: Il numero degl

}

// Verifica se l'intervallo è valido

if (lower >= upper) {

throw new IllegalArgumentException("\n->Errore: Limite inferio

}

// Verifica se il numero di numeri è positivo

if (count <= 0) {

throw new IllegalArgumentException("\n->Errore: Il numero di e

}

Set<Integer> uniqueNumbers = new HashSet<>();

Coverage

RandomNumberGeneratorAutomateTestCase

Element

Class, %

Method, %

Line, %

org

100% (1/1)

100% (1/1)

100% (19/19)

example

100% (1/1)

100% (1/1)

100% (19/19)

RandomNumberGenerator

100% (1/1)

100% (1/1)

100% (19/19)

Test Results

137 ms

Tests passed: 21 of 21 tests - 137 ms

Test class RandomNumberGeneratorAutomateTestCase

137 ms

Task: compileJava UP-TO-DATE

omNumberGenerator > src > main > java > org > example > RandomNumberGenerator

9:14

LF

UTF-8

4 space



```
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test

---- IntelliJ IDEA coverage runner ----
Line coverage ...
include patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.602808s for 1695 classes or 3.5563893805309736E-4s per class

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they
are expected. For more on this, please refer to https://docs.gradle.org/8.2/userguide/command\_line\_interface.html#sec:command\_line\_warnings.

BUILD SUCCESSFUL in 4s
3 actionable tasks: 2 executed, 1 up-to-date
09:51:44: Execution finished ':test --tests "RandomNumberGeneratorAutomateTestCase"'.

```

Test class RandomNumberGeneratorAutomateTestCase	224 ms
threeElementsStringToIntWithValidIntervalAndNOTValidNumCountThrowsException	162 ms
[1] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = -5	passed 153 ms
[2] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = 0	passed 7 ms
[3] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = 11	passed 2 ms
oneElementNullOrEmptyThrowsException	11 ms
[1] LOW LIMIT = null, UP LIMIT = 10, #ELEMENTS = 5	passed 2 ms
[2] LOW LIMIT = null, UP LIMIT = 10, #ELEMENTS = 5	passed 1 ms
[3] LOW LIMIT = 1, UP LIMIT = null, #ELEMENTS = 5	passed 4 ms
[4] LOW LIMIT = 1, UP LIMIT = null, #ELEMENTS = 5	passed 2 ms
[5] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = null	passed 1 ms
[6] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = null	passed 1 ms
threeElementsStringToIntWithNOTValidIntervalAndValidNumCountThrowsException	11 ms
[1] LOW LIMIT = 10, UP LIMIT = 1, #ELEMENTS = 5	passed 1 ms
[2] LOW LIMIT = 10, UP LIMIT = 10, #ELEMENTS = 1	passed 10 ms
threeElementsStringToIntWithValidIntervalAndValidNumCountReturn SortedArray	passed 15 ms
oneElementStringToIntThrowsException	5 ms
[1] LOW LIMIT = a, UP LIMIT = 2147483648, #ELEMENTS = 5	passed 2 ms
[2] LOW LIMIT = a, UP LIMIT = 10, #ELEMENTS = %	passed 2 ms
[3] LOW LIMIT = 1, UP LIMIT = 2147483648, #ELEMENTS = @	passed 1 ms
twoElementStringToIntThrowsException	9 ms
[1] LOW LIMIT = a, UP LIMIT = 10, #ELEMENTS = 5	passed 6 ms
[2] LOW LIMIT = 1, UP LIMIT = 2147483648, #ELEMENTS = 5	passed 2 ms
[3] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = @	passed 1 ms
whiteSpaceInOneElementThrowsException	11 ms
[1] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = 5	passed 5 ms
[2] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = 5	passed 2 ms
[3] LOW LIMIT = 1, UP LIMIT = 10, #ELEMENTS = 5	passed 4 ms

4: Coprire parti mancanti

Se questi pezzi di codice devono essere testati, implementare il test e tornare al punto 3

FINE

Homework 1

PROPERTY BASED TESTING

```
package org.example;
```

```
// Press Shift twice to open the Search Everywhere dialog and type `show whitespaces`,
// then press Enter. You can now see whitespace characters in your code.
```

```
import java.util.Random;
import java.util.Scanner;
import java.util.*;
```

7 usages donato_santacesaria *

```
public class RandomNumberGenerator {
```

6 usages donato_santacesaria *

```
public static int[] generateUniqueRandomNumbersInRange(String lowerLimit, String upperLimit, String numCount)
    throws IllegalArgumentException {
```

```
    try {
```

```
        // Conversione degli operandi in interi
        int lower = Integer.parseInt(lowerLimit);
        int upper = Integer.parseInt(upperLimit);
        int count = Integer.parseInt(numCount);
```

```
        if((upper-lower+1)<count){
```

```
            throw new IllegalArgumentException("\n->Errore: Il numero degli elementi da generare deve essere al massimo pari alla differenza+1 tra il limite superiore ed il
```

```
        // Verifica se l'intervallo è valido
```

```
        if (lower >= upper) {
```

```
            throw new IllegalArgumentException("\n->Errore: Limite inferiore deve essere inferiore al limite superiore.");
        }
```

```
        // Verifica se il numero di numeri è positivo
```

```
        if (count <= 0) {
```

```
            throw new IllegalArgumentException("\n->Errore: Il numero di elementi deve essere positivo.");
        }
```

```
        Set<Integer> uniqueNumbers = new HashSet<>();
```

```
        Random random = new Random();
```

```

16 int lower = Integer.parseInt(lowerLimit);
17 int upper = Integer.parseInt(upperLimit);
18 int count = Integer.parseInt(numCount);
19
20 if((upper-lower+1)<count){
21     throw new IllegalArgumentException("\n->Errore: Il numero degli elementi da generare deve essere al massimo pari alla differenza+1 tra il limite superiore ed il
22 }
23
24 // Verifica se l'intervallo è valido
25 if (lower >= upper) {
26     throw new IllegalArgumentException("\n->Errore: Limite inferiore deve essere inferiore al limite superiore.");
27 }
28
29 // Verifica se il numero di numeri è positivo
30 if (count <= 0) {
31     throw new IllegalArgumentException("\n->Errore: Il numero di elementi deve essere positivo.");
32 }
33
34 Set<Integer> uniqueNumbers = new HashSet<>();
35 Random random = new Random();
36
37 // Genera numeri unici nell'intervallo
38 while (uniqueNumbers.size() < count) {
39     int randomNumber = random.nextInt( bound: upper - lower + 1) + lower;
40     uniqueNumbers.add(randomNumber);
41 }
42
43 // Converti il set in un array ordinato
44 int[] result = uniqueNumbers.stream().sorted().mapToInt(Integer::intValue).toArray();
45
46 return result;
47
48 } catch (NumberFormatException e) {
49     throw new IllegalArgumentException("\n->Errore: Inserito un valore non valido. Assicurati di inserire valori validi.");
50 }
51 }
52 }
53

```


1: Esprimere le proprietà da testare

Definire un insieme di proprietà che il nostro programma dovrebbe rispettare

1 . Proprietà relative gli input

- a) Se anche uno solo degli **input** passati **non è valido** (i.e. non è una stringa che rappresenta un intero), il programma deve **lanciare un'eccezione**;
- b) Se l'**intervallo** passato **non è valido** (i.e. il limite inferiore è maggiore del limite superiore), il programma deve **lanciare un'eccezione**;
- c) Se il **numero di elementi** da generare **non è valido** (i.e. minore o uguale a zero o superiore alla dimensione+1 dell'intervallo), il programma deve **lanciare un'eccezione**;

2. Proprietà relative l'output

- a) L'output generato deve essere un **array di dimensione pari al numero di elementi da generare** passato al programma;
- b) Gli **elementi** dell'array devono essere **interni all'intervallo** passato al programma;
- c) Gli **elementi** dell'array generato devono essere **unici**;
- d) Gli **elementi** dell'array generato devono essere **ordinati in modo crescente**.

2: Lasciare che sia il framework a scegliere i test

Il framework chiama il metodo sotto test più volte con diversi parametri di input

1 . a) Se anche uno solo degli input passati non è valido (i.e. non è una stringa che rappresenta un intero), il programma deve lanciare un'eccezione

```
no usages new *
@Provide
Arbitrary<String> nonIntString() {
    return Arbitraries.strings().ascii().filter(s -> !s.matches(regex: "\\d+"));
}
```

I. Limite inferiore **non valido**

```
/*1.a).I Limite inferiore non valido; */
new *
@Property(generation = GenerationMode.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void invalidLowerLimitThrowsException(
    @ForAll("nonIntString") String lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange (min=0, max=4000000)int numCount
) {
    assertThrows(IllegalArgumentException.class, () -> {
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(lowerLimit, String.valueOf(upperLimit), String.valueOf(numCount));
    });
    Statistics.collect(lowerLimit);
}
```

1 . a) . I- statistiche

Nei test 1.a) ci limitiamo ad osservare i valori generati da jqwik rispetto alle caratteristiche attese.

In questo caso osserviamo i valori non numerici assunti da lowerLimit. Notiamo che jqwik ha concentrato i valori generati in quei singoli caratteri ASCII che precedono e seguono i caratteri ASCII che rappresentano numeri.

```
timestamp = 2023-12-19T15:52:26.601804, PropertyBasedTest:invalidLowerLimitThrowsException =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 1000 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 100 | # of all combined edge cases
edge-cases#tried = 100 | # of edge cases tried in current run
seed = 3336337938711337238 | random seed to reproduce generated values
```

```
timestamp = 2023-12-19T15:52:26.592621, [PropertyBasedTest:invalidLowerLimitThrowsException] (1000) statistics =
```

```
# |
| label | count |
-----|-----|-----
0 |
| 61 | ?????????????????????????????????????????????????????????????
1 |
| 37 | ?????????????????????????????????????????????????????????????
2 |
| 1 | ?
3 |
| 1 | ?
4 |
| 1 | ?
5 |
| 1 | ?
6 |
| 1 | ?
7 |
| 1 | ?
8 |
| 1 | ?
9 |
| 1 | ?
```

1 . a) Se anche uno solo degli input passati non è valido (i.e. non è una stringa che rappresenta un intero), il programma deve lanciare un'eccezione

```
no usages new *
@Provide
Arbitrary<String> nonIntString() {
    return Arbitraries.strings().ascii().filter(s -> !s.matches(regex: "\\d+"));
}
```

II. Limite superiore **non valido**

```
/*1.a).II limite superiore non valido; */
new *
@Property(generation = GenerationMode.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void invalidUpperLimitThrowsException(
    @ForAll @IntRange int lowerLimit,
    @ForAll ("nonIntString") String upperLimit,
    @ForAll @IntRange (min=0, max=40000000)int numCount
) {
    assertThrows(IllegalArgumentException.class, () -> {
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(String.valueOf(lowerLimit),upperLimit, String.valueOf(numCount));
    });
    Statistics.collect(upperLimit);
}
```

1 . a) . II- statistiche

Nei test 1.a) ci limitiamo ad osservare i valori generati da jqwik rispetto alle caratteristiche attese.

In questo caso osserviamo i valori non numerici assunti da upperLimit. Notiamo che jqwik ha concentrato i valori generati in quei singoli caratteri ASCII che precedono e seguono i caratteri ASCII che rappresentano numeri.

```
timestamp = 2023-12-19T15:52:25.822036100, PropertyBasedTest:invalidUpperLimitThrowsException =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 1000 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 100 | # of all combined edge cases
edge-cases#tried = 82 | # of edge cases tried in current run
seed = 684529996837039484 | random seed to reproduce generated values
```

```
timestamp = 2023-12-19T15:52:25.718737, [PropertyBasedTest:invalidUpperLimitThrowsException] (1000) statistics =
```


1 . a) Se anche uno solo degli input passati non è valido (i.e. non è una stringa che rappresenta un intero), il programma deve lanciare un'eccezione

```
no usages new *
@Provide
Arbitrary<String> nonIntString() {
    return Arbitraries.strings().ascii().filter(s -> !s.matches(regex: "\\d+"));
}
```

III. Numero di elementi da generare **non valido**

```
/*1.a).III Numero di elementi da generare non valido; */
new *
@property(generation = GenerationMode.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void invalidNumCountThrowsException(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll ("nonIntString") String numCount
) {
    assertThrows(IllegalArgumentException.class, () -> {
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(String.valueOf(lowerLimit), String.valueOf(upperLimit), numCount);
    });
    Statistics.collect(numCount);
}
```

1. a) . III- statistische

Nei test 1.a) ci limitiamo ad osservare i valori generati da jqwik rispetto alle caratteristiche attese.

In questo caso osserviamo i valori non numerici assunti da numCount. Notiamo che jqwik ha concentrato i valori generati in quei singoli caratteri ASCII che precedono e seguono i caratteri ASCII che rappresentano numeri.

```
timestamp = 2023-12-19T15:52:26.149333800, PropertyBasedTest:invalidNumCountThrowsException =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 1000 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 100 | # of all combined edge cases
edge-cases#tried = 77 | # of edge cases tried in current run
seed = 3833411614195859447 | random seed to reproduce generated values
```

```
timestamp = 2023-12-19T15:52:26.107352400, [PropertyBasedTest:invalidNumCountThrowsException] (1000) statistics =
```

1 . b) Se l'intervallo passato non è valido (i.e. il limite inferiore è maggiore del limite superiore), il programma deve lanciare un'eccezione;

```
/*1.b) Se l'intervallo passato non è valido (i.e. il limite inferiore è maggiore del limite superiore),
il programma deve lanciare un'eccezione; */
new *
@property(generation = GenerationType.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void invalidIntervalThrowsException(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange int numCount
){
    Assume.that( condition: lowerLimit > upperLimit);
    assertThrows(IllegalArgumentException.class, () -> {
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(String.valueOf(lowerLimit), String.valueOf(upperLimit), String.valueOf(numCount));
    });
    String range=lowerLimit<=upperLimit+1?"test su valori boundary vicini ad upperLimit":
        "test su altri scenari";
    Statistics.label("range").collect(range);
    Statistics.label("value").collect(lowerLimit, upperLimit);
}
```

1 . b) - statistiche

Verifichiamo se jqwik genera valori di lowerLimit e UpperLimit utili a creare eccezioni ed eseguire il test. In quasi il 50% dei casi (486), jqwik ha successo. In termini assoluti, i test realizzati con i valori utilizzabili generati sono in 16 casi test su valori boundary rispetto ad upperLimit, in 470 casi sono test su altri scenari.

```
timestamp = 2023-12-19T15:52:26.419847500, PropertyBasedTest:invalidIntervalThrowsException =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 486 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 125 | # of all combined edge cases
edge-cases#tried = 118 | # of edge cases tried in current run
seed = 88487082226341589 | random seed to reproduce generated values
```

```
timestamp = 2023-12-19T15:52:26.385331400, [PropertyBasedTest:invalidIntervalThrowsException] (486) range =
```

#	label	count
0	test su altri scenari	470
1	test su valori boundary vicini ad upperLimit	16

```
timestamp = 2023-12-19T15:52:26.405367600, [PropertyBasedTest:invalidIntervalThrowsException] (486) value =
```

#	label	count
0	1 0	5
1	2 0	7
2	2 1	5
3	4 2	1
4	9 6	1
5	12 0	1
6	12 1	1
7	13 2	1
8	16 2	1
9	17 8	1
10	17 12	1
11	18 0	2
12	18 6	1
13	19 2	1
14	20 2	1
15	21 0	1

1 . c) Se il numero di elementi da generare non è valido (i.e. minore o uguale a zero o superiore alla dimensione+1 dell'intervallo), il programma deve lanciare un'eccezione;

```
/*1.c) Se il numero di elementi da generare non è valido (i.e. minore o uguale a zero o superiore alla dimensione+1 dell'intervallo),
il programma deve lanciare un'eccezione; */
new *
@property(generation = GenerationMode.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void invalidElementsToGenerateThrowsException(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange int numCount
){
    Assume.that( condition: lowerLimit < upperLimit);
    Assume.that( condition: numCount <= 0 || numCount > (upperLimit - lowerLimit + 1));

    assertThrows(IllegalArgumentException.class, () -> {
        RandomNumberGenerator.generateUniqueRandomNumbersInRange(String.valueOf(lowerLimit),
            String.valueOf(upperLimit), String.valueOf(numCount));
    });

    String range;
    if (numCount == 0) {
        range = "test su valori boundary vicini a 0";
    } else if (numCount == (upperLimit - lowerLimit + 2)) {
        range = "test su valori boundary vicini a dimensione intervallo";
    } else {
        range = "test su altri valori";
    }
    Statistics.label("range").collect(range);
    Statistics.label("value").collect(lowerLimit, upperLimit, numCount);
}
```

1 . c) - statistiche

Verifichiamo se jqwik genera valori di numCount in grado di generare eccezioni quindi eseguire il test. Ha successo in circa il 18% dei casi. I test eseguiti riguardano nella maggior parte dei casi scenari non boundary.

```
timestamp = 2023-12-19T16:36:45.382681300, PropertyBasedTest:invalidElementsToGenerateThrowsException =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 184 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 125 | # of all combined edge cases
edge-cases#tried = 107 | # of edge cases tried in current run
seed = -7364670978066650558 | random seed to reproduce generated values
```

```
timestamp = 2023-12-19T16:36:45.377160200, [PropertyBasedTest:invalidElementsToGenerateThrowsException] (184) range =
# | label | count |
----|-----|-----|
0 | test su altri valori | 169 | ?????????????????????????????????????????????????????????????
1 | test su valori boundary vicini a 0 | 12 | ?????
2 | test su valori boundary vicini a dimensione intervallo | 3 | ?
```

```
timestamp = 2023-12-19T16:36:45.381683700, [PropertyBasedTest:invalidElementsToGenerateThrowsException] (184) value =
# | label | count |
----|-----|-----|
0 | 0 5719 352806 | 1 | ?
1 | 0 2147483646 0 | 1 | ?
2 | 0 44 2950 | 1 | ?
3 | 0 11749 124959 | 1 | ?
4 | 0 2147483647 2147483646 | 1 | ?
5 | 0 2147483647 2147483647 | 1 | ?
6 | 0 2147483647 0 | 1 | ?
7 | 0 2147483647 1 | 1 | ?
8 | 0 2147483647 2 | 1 | ?
9 | 0 71 131 | 1 | ?
10 | 0 1 2147483646 | 1 | ?
11 | 0 1 2147483647 | 1 | ?
12 | 0 1 0 | 1 | ?
13 | 0 311 16257 | 1 | ?
14 | 0 2 2147483646 | 1 | ?
15 | 0 2 2147483647 | 1 | ?
```

2. a) L'output generato deve essere un array di dimensione pari al numero di elementi da generare passato al programma;

```
new *
@property(generation = GenerationMode.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void generatedArrayShouldHaveLengthDefinedByTheUser(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange (max=4000000)int numCount //si imposta un limite superiore per evitare errori di memoria/tempi di esecuzione
) {
    // si assicura che l'intervallo che il numero di elementi da generare siano validi
    Assume.that( condition: lowerLimit < upperLimit);
    Assume.that( condition: numCount>0 && numCount<=(upperLimit-lowerLimit+1));

    int[] result = RandomNumberGenerator.generateUniqueRandomNumbersInRange(
        String.valueOf(lowerLimit),
        String.valueOf(upperLimit),
        String.valueOf(numCount)
    );
    assertEquals(numCount, result.length, message: "La lunghezza dell'array non è uguale al numero specificato");
    Statistics.collect(result.length, numCount);

    String range;
    if (numCount == 1) {
        range = "test su valori boundary di NumCount vicini a 0";
    } else if (numCount == (upperLimit - lowerLimit+1)) {
        range = "test su valori boundary di NumCount vicini a dimensione intervallo";
    } else if (lowerLimit==(upperLimit-1)){
        range="test su valori boundary dell'intervallo";
    }
    else {
        range = "test su altri valori";
    }
    Statistics.label("range").collect(range);
    Statistics.label("value").collect(lowerLimit, upperLimit, numCount);
}
```

2. a) - statistiche

Verifichiamo se jqwik è in grado di generare test che rispettino le condizioni di esecuzione del programma, evitando le eccezioni. Si osservano i valori della lunghezza degli array generati uguali al NumCount passato. Si osservano poi che tipi di test siano stati prodotti, distinguendo in: boundary di NumCount vicini a 0; e boundary sulla validità del limite; altri scenari.

```
timestamp = 2023-12-20T08:08:23.894076300, PropertyBasedTest:generatedArrayShouldHaveLengthDefinedByTheUser =
|-----jqwik-----
tries = 1000      | # of calls to property
checks = 323     | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 125 | # of all combined edge cases
edge-cases#tried = 117 | # of edge cases tried in current run
seed = 5583511567722096579 | random seed to reproduce generated values
```

```
timestamp = 2023-12-20T08:08:23.880573100, [PropertyBasedTest:generatedArrayShouldHaveLengthDefinedByTheUser] (323) statistics =
```

#	label	count	
0	1 1	18	????????????????
1	2 2	17	????????????????
2	3 3	2	??
3	4 4	2	??
4	6 6	1	?
5	7 7	2	??
6	8 8	1	?

```
timestamp = 2023-12-20T08:08:23.881694600, [PropertyBasedTest:generatedArrayShouldHaveLengthDefinedByTheUser] (323) range =
```

#	label	count	
0	test su altri valori	302	??
1	test su valori boundary di NumCount vicini a 0	18	????
2	test su valori boundary di NumCount vicini a dimensione intervallo	3	

```
timestamp = 2023-12-20T08:08:23.887199200, [PropertyBasedTest:generatedArrayShouldHaveLengthDefinedByTheUser] (323) value =
```

#	label	count	
0	0 5112511 29680	1	?
1	0 2147483646 1	1	?
2	0 2147483646 0	1	?

2. b) Gli elementi dell'array devono essere interni all'intervallo passato al programma;

```
/*2.b) Gli elementi dell'array devono essere interni all'intervallo passato al programma; */
new *
@property(generation = GenerationType.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void generatedArrayShouldBeWithinSpecifiedRange(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange (max=4000000)int numCount //si imposta un limite superiore per evitare errori di memoria
) {
    // si assicura che l'intervallo che il numero di elementi da generare siano validi
    Assume.that( condition: lowerLimit < upperLimit);
    Assume.that( condition: numCount > 0 && numCount < (upperLimit - lowerLimit + 1));

    int[] result = RandomNumberGenerator.generateUniqueRandomNumbersInRange(
        String.valueOf(lowerLimit),
        String.valueOf(upperLimit),
        String.valueOf(numCount)
    );
    // Verifica che tutti gli elementi siano nell'intervallo specificato
    for (int number : result) {
        assertTrue( condition: number >= lowerLimit && number <= upperLimit,
            message: "Elemento dell'array non è nell'intervallo specificato");
    }
    String range="";
    for (int number:result){
        if(number==lowerLimit){
            range = "test su valori boundary sul limite inferiore";
        } else if ((number==upperLimit)) {
            range = "test su valori boundary sul limite superiore";
        }else{
            range = "test su valori boundary su altri valori";
        }
    }
    Statistics.label("range").collect(range);
    Statistics.label("value").collect(lowerLimit, upperLimit);
}
```

2. b) - statistiche

Verifichiamo se jqwik è in grado di generare test che rispettino le condizioni di esecuzione del programma, evitando le eccezioni. Si osservano i valori dei limiti inferiore e superiore generati. Si osservano poi che tipi di test siano stati prodotti, distinguendo in: boundary sul limite inferiore; boundary limite superiore; altri scenari.

```
timestamp = 2023-12-20T08:33:42.841400700, PropertyBasedTest:generatedArrayShouldBeWithinSpecifiedRange =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 332 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 125 | # of all combined edge cases
edge-cases#tried = 121 | # of edge cases tried in current run
seed = -2990763392420083846 | random seed to reproduce generated values
```

```
timestamp = 2023-12-20T08:33:42.836968100, [PropertyBasedTest:generatedArrayShouldBeWithinSpecifiedRange] (332) range =
```

#	label	count
0	test su valori boundary su altri valori	302
1	test su valori boundary sul limite inferiore	1
2	test su valori boundary sul limite superiore	29

```
timestamp = 2023-12-20T08:33:42.840400300, [PropertyBasedTest:generatedArrayShouldBeWithinSpecifiedRange] (332) value =
```

#	label	count
0	0 2147483646	4
1	0 2	2
2	0 3723	1
3	0 576700	1
4	0 61800264	1
5	0 4865	1
6	0 1825	1
7	0 13111	1

2. c) Gli elementi dell'array generato devono essere unici;

```
/*2.c) Gli elementi dell'array devono essere unici; */
new *
@property(generation = GenerationMode.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void generatedArrayShouldHaveUniqueElements(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange (max=4000000)int numCount //si imposta un limite superiore per evitare errori di memoria,
) {
    // si assicura che l'intervallo che il numero di elementi da generare siano validi
    Assume.that( condition: lowerLimit < upperLimit);
    Assume.that( condition: numCount > 0 && numCount < (upperLimit - lowerLimit + 1));
    int[] result = RandomNumberGenerator.generateUniqueRandomNumbersInRange(
        String.valueOf(lowerLimit),
        String.valueOf(upperLimit),
        String.valueOf(numCount)
    );
    // si verifica che non ci siano elementi duplicati nell'array
    Set<Integer> uniqueElements = new HashSet<>();
    for (int number : result) {
        assertFalse(uniqueElements.contains(number),
            message: "Elementi duplicati nell'array generato");
        uniqueElements.add(number);
    }

    String range="";
    if(numCount<100){
        range="Test effettuati su un numero basso di valori";
    } else if (numCount>=100&&numCount<3000000) {
        range="Test effettuati su un numero medio di valori";
    } else if (numCount>=3000000) {
        range="Test effettuati su un numero alto di valori";
    }
    Statistics.label("range").collect(range);
    Statistics.label("value").collect(result.length);
}
```

2. c) - statistiche

Verifichiamo se jqwik è in grado di generare test che rispettino le condizioni di esecuzione del programma, evitando le eccezioni. Osserviamo su quali tipi di test è stata verificata la mancanza di duplicazioni: se su test composti da un numero basso di valori, se su test composti da un numero medio di valori o se su test composti da un numero elevato di valori. Quindi si dà evidenza alle dimensioni osservate nei test.

```
timestamp = 2023-12-20T08:49:01.223648800, PropertyBasedTest:generatedArrayShouldHaveUniqueElements =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 334 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 125 | # of all combined edge cases
edge-cases#tried = 123 | # of edge cases tried in current run
seed = 2698634896610274900 | random seed to reproduce generated values
```

```
timestamp = 2023-12-20T08:49:01.217280400, [PropertyBasedTest:generatedArrayShouldHaveUniqueElements] (334) range =
# | label | count |
-----|-----|-----|-----
0 | Test effettuati su un numero alto di valori | 19 | ???????
1 | Test effettuati su un numero basso di valori | 110 | ?????????????????????????????????????????????????????????????
2 | Test effettuati su un numero medio di valori | 205 | ?????????????????????????????????????????????????????????????
```

```
timestamp = 2023-12-20T08:49:01.222651300, [PropertyBasedTest:generatedArrayShouldHaveUniqueElements] (334) value =
# | label | count |
-----|-----|-----|-----
0 | 1 | 15 | ???????????????
1 | 2 | 12 | ??????????????
2 | 4 | 3 | ???
3 | 5 | 6 | ??????
4 | 6 | 4 | ????
5 | 7 | 2 | ??
6 | 8 | 5 | ?????
7 | 9 | 1 | ?
```

2. d) Gli elementi dell'array generato devono essere ordinati in modo crescente.

```
/*2.d) Gli elementi dell'array devono essere ordinati in modo crescente; */
new *
@property(generation = GenerationType.RANDOMIZED)
@StatisticsReport(format = Histogram.class)
void generatedArrayShouldBeSortedInAscendingOrder(
    @ForAll @IntRange int lowerLimit,
    @ForAll @IntRange int upperLimit,
    @ForAll @IntRange(max = 4000000) int numCount // si imposta un limite superiore per evitare errori di memoria
) {
    // si assicura che l'intervallo che il numero di elementi da generare siano validi
    Assume.that( condition: lowerLimit < upperLimit);
    Assume.that( condition: numCount > 0 && numCount < (upperLimit - lowerLimit + 1));
    int[] result = RandomNumberGenerator.generateUniqueRandomNumbersInRange(
        String.valueOf(lowerLimit),
        String.valueOf(upperLimit),
        String.valueOf(numCount)
    );
    // Verifica che gli elementi siano ordinati in modo crescente
    for (int i = 1; i < result.length; i++) {
        assertTrue( condition: result[i - 1] <= result[i],
            message: "Gli elementi dell'array non sono ordinati in modo crescente");
    }
    String range="";
    if(numCount<100){
        range="Test effettuati su un numero basso di valori";
    } else if (numCount>=100&&numCount<3000000) {
        range="Test effettuati su un numero medio di valori";
    } else if (numCount>=3000000) {
        range="Test effettuati su un numero alto di valori";
    }
    Statistics.label("range").collect(range);
    Statistics.label("value").collect(result.length);
}
```

2. d) - statistiche

Verifichiamo se jqwik è in grado di generare test che rispettino le condizioni di esecuzione del programma, evitando le eccezioni. Osserviamo su quali tipi di test è stato verificato l'ordinamento crescente: se su test composti da un numero basso di valori, se su test composti da un numero medio di valori o se su test composti da un numero elevato di valori. Quindi si dà evidenza alle dimensioni osservate nei test.

```
timestamp = 2023-12-20T09:00:49.067243200, PropertyBasedTest:generatedArrayShouldBeSortedInAscendingOrder =
|-----jqwik-----
tries = 1000 | # of calls to property
checks = 333 | # of not rejected calls
generation = RANDOMIZED | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW | fixing the random seed is allowed
edge-cases#mode = MIXIN | edge cases are mixed in
edge-cases#total = 125 | # of all combined edge cases
edge-cases#tried = 110 | # of edge cases tried in current run
seed = 1476742506910574633 | random seed to reproduce generated values
```

```
timestamp = 2023-12-20T09:00:49.064720, [PropertyBasedTest:generatedArrayShouldBeSortedInAscendingOrder] (333) range =
# | label | count |
-----|-----|-----|
0 | Test effettuati su un numero alto di valori | 17 | ?????
1 | Test effettuati su un numero basso di valori | 106 | ?????????????????????????????????????????
2 | Test effettuati su un numero medio di valori | 210 | ?????????????????????????????????????????
```

```
timestamp = 2023-12-20T09:00:49.067243200, [PropertyBasedTest:generatedArrayShouldBeSortedInAscendingOrder] (333) value =
```

```
# | label | count |
-----|-----|-----|
0 | 1 | 15 | ?????????????
1 | 2 | 16 | ?????????????
2 | 3 | 5 | ?????
3 | 4 | 5 | ?????
4 | 5 | 2 | ??
5 | 7 | 5 | ?????
6 | 8 | 1 | ?
7 | 9 | 4 | ?????
8 | 10 | 3 | ???
9 | 11 | 2 | ??
10 | 12 | 1 | ?
11 | 13 | 1 | ?
12 | 14 | 4 | ???
13 | 15 | 1 | ?
```

[Collapse](#) | [Expand](#)

Test class PropertyBasedTest

5 m 59 s

invalidUpperLimitThrowsException	passed	1.26 s
invalidNumCountThrowsException	passed	431 ms
invalidIntervalThrowsException	passed	312 ms
generatedArrayShouldHaveUniqueElements	passed	1 m 43 s
invalidLowerLimitThrowsException	passed	763 ms
generatedArrayShouldBeWithinSpecifiedRange	passed	1 m 25 s
invalidElementsToGenerateThrowsException	passed	304 ms
generatedArrayShouldHaveLengthDefinedByTheUser	passed	1 m 31 s
generatedArrayShouldBeSortedInAscendingOrder	passed	1 m 16 s

Generated by IntelliJ IDEA on 31/12/23, 11:27

Coverage PropertyBasedTest ×



Element ^	Class, %	Method, %	Line, %
org.example	100% (1/1)	100% (1/1)	100% (19/19)
RandomNumberGenerator	100% (1/1)	100% (1/1)	100% (19/19)

FINE

Integrazione e Test di Sistemi Software

SNTDNT

Donato Santacesaria

Matricola 754727